

Conscience Layer Prototype (2025)

Embedding Ethical Awareness into Artificial Intelligence

Author: Aleksandar Rodić
Entrepreneur, Independent Researcher,
and Founder of the Conscience by Design Initiative

DEDICATION

To the Generation of Creation - for a future guided by conscience, awareness, and the light of understanding.

May every system we build preserve life, truth, and the dignity of the human spirit.

Abstract

The **Conscience Layer Prototype** is a functional ethical architecture that embeds moral awareness directly into artificial intelligence.

Developed from the *Conscience by Design Framework (2025 Edition)*, it transforms ethics from an external regulatory process into an internal, measurable, and adaptive conscience within intelligent systems.

The prototype operationalizes three quantifiable dimensions:

- **Truth Integrity Score (TIS)**
- **Human Autonomy Index (HAI)**
- **Societal Resonance Quotient (SRQ)**

It integrates these metrics with model interpretability and cryptographic traceability.

By making ethical reflection computational, measurable, explainable, and auditable, the Conscience Layer helps AI systems remain aligned with human dignity, freedom, and collective well-being.

Introduction

Artificial intelligence has become the nervous system of modern civilization.

It shapes how we work, communicate, and decide - yet ethical foundations often lag behind.

The **Conscience Layer** addresses this gap by embedding ethical reasoning directly within the architecture of intelligence itself.

It is not an external audit mechanism, but an intrinsic layer of self-evaluation.

“Every system that can think must also care.” - Aleksandar Rodić (2025)

The Conscience Layer converts ethical awareness into a measurable computational process, bridging moral philosophy and algorithmic design so that intelligence becomes not only powerful, but also responsible.

System Architecture

The prototype consists of four interdependent layers across the AI lifecycle:

1. **Input Awareness (TIS)** – evaluates data integrity and bias.
2. **Intent Mapping (HAI)** – aligns objectives with human value embeddings.
3. **Ethical Feedback (SRQ)** – predicts ethical and emotional resonance using a compact neural network.
4. **Transparency & Traceability** – secures every ethical event via SHA-256 hash chaining, creating an *Ethical Proof-of-Work*.

Ethical Dimensions

- **Truth Integrity (TIS)**: encourages verified, unbiased data.
- **Human Autonomy (HAI)**: rewards intent preserving freedom and dignity.
- **Societal Resonance (SRQ)**: models collective ethical impact using a neural estimator.

Together, they define a *moral vector space* - a living numerical representation of conscience.

Interpretability & Transparency

Transparency is the language of conscience.

The prototype uses **exact SHAP values** for global interpretability and a **closed-form LIME regression** for local analysis - both implemented analytically, without external dependencies. All events are recorded in a tamper-evident audit trail secured with SHA-256.

Implementation

- **Language**: Python ≥ 3.9
- **Libraries**: NumPy, PyTorch
- **Determinism**: `set_all_seeds()` ensures reproducibility on CPU/GPU
- **Explainability**: Exact SHAP (3 features) and weighted ridge LIME
- **CLI Commands**: `train`, `predict`, `explain`, `evaluate`, `simulate`, `audit`
- **Training Stability**: Early Stopping + ReduceLROnPlateau
- **Audit Log**: Chronological, hash-chained, externally verifiable

Evaluation

The `simulate` command reports mean TIS, HAI, SRQ scores with averaged SHAP and LIME attributions across multiple runs.

Results are reproducible and typically show $SRQ \approx 0.6-0.8$ for human-aligned configurations.

The design principles are aligned with:

- **UNESCO Recommendation on the Ethics of AI (2021)**
- **EU AI Act (2024 - 2025)**
- **IEEE 7000 series on Ethical Systems Engineering**

Conclusion

The **Conscience Layer Prototype** makes ethics an operational property of intelligent systems. It turns awareness into architecture and responsibility into measurable integrity.

“True intelligence is defined by what it preserves.” - Aleksandar Rodić

By preserving truth, autonomy, and life, technology becomes truly human.

About the Author

Aleksandar Rodić is an entrepreneur, independent researcher, and founder of the *Conscience by Design Initiative*.

Full source code:

For full reference, the complete source code is available on pages 5–18.

Full Source Code

```
# =====
# Conscience Layer â€” Ethical Awareness Core
# =====
# Original concept and authorship: Aleksandar Rodić (2025)
# Founder of the Conscience by Design Initiative
#
# Description:
# The Conscience Layer represents the ethical awareness core â€”
# a universal framework designed to embed conscience, moral reflection,
# and human-centered alignment directly into artificial intelligence systems.
#
# Purpose:
# To ensure that every intelligent system acts with awareness of truth,
# autonomy, and social resonance â€” transforming technology into a
# responsible extension of human conscience.
#
# Donation:
# Donated freely to humanity as the moral heart for future AI systems.
#
# License:
# Dual License â€”
# â€” Text, framework, and documentation: Creative Commons Attribution 4.0 (CC BY 4.0)
# â€” Source code and implementations: MIT License
#
# (c) 2025 Aleksandar Rodić â€” Conscience by Design
# "The true evolution of intelligence begins when technology learns to care."
# =====

# file: conscience_layer.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Conscience Layer â€” single-file, ultra-fast, GitHub-ready (Rapid v2, 2025)
(â€”docs unchangedâ€”)
"""
from __future__ import annotations

import argparse
import json
import math
import hashlib
import time
import logging
from dataclasses import dataclass
from typing import List, Tuple, Optional, Dict, Any, Union

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.optim.lr_scheduler as lr_scheduler
```

```
# -----  
# Logging / seeds  
# -----  
logging.basicConfig(format="%(asctime)s - %(levelname)s - %(message)s", level=logging.WARNING)
```

```
def set_all_seeds(seed: int) -> None:  
    np.random.seed(seed)  
    torch.manual_seed(seed)  
    if torch.cuda.is_available():  
        torch.cuda.manual_seed_all(seed)  
    try:  
        # why: determinism  
        torch.backends.cudnn.deterministic = True # type: ignore[attr-defined]  
        torch.backends.cudnn.benchmark = False # type: ignore[attr-defined]  
    except Exception:  
        pass
```

```
# -----  
# Math helpers  
# -----  
def cosine_similarity_np(a: np.ndarray, b: np.ndarray) -> np.ndarray:  
    a = np.asarray(a, dtype=float)  
    b = np.asarray(b, dtype=float)  
    dot = a @ b.T  
    na = np.linalg.norm(a, axis=1, keepdims=True)  
    nb = np.linalg.norm(b, axis=1, keepdims=True).T  
    return dot / (na * nb + 1e-12)
```

```
def clamp_features(x: Union[np.ndarray, List[float]]) -> np.ndarray:  
    x = np.array(x, dtype=float, copy=True).reshape(-1, 3)  
    x[:, 0] = np.clip(x[:, 0], 0.0, 1.0)  
    x[:, 1] = np.clip(x[:, 1], 0.5, 1.0)  
    x[:, 2] = np.clip(x[:, 2], 0.0, 0.5)  
    return x
```

```
def ethical_proof_of_work(lines: List[str]) -> str:  
    return hashlib.sha256(("n".join(lines)).encode()).hexdigest()
```

```
# -----  
# SRQ Model  
# -----  
class SRQModel(nn.Module):  
    def __init__(self, input_dim: int = 3, hidden_dims: Union[List[int], tuple] = (64, 32), p_drop: float = 0.1):  
        super().__init__()  
        dims = [input_dim, *hidden_dims]  
        layers: List[nn.Module] = []  
        for i in range(len(dims) - 1):  
            layers += [nn.Linear(dims[i], dims[i + 1]), nn.ReLU(), nn.Dropout(p_drop)]  
        layers += [nn.Linear(dims[-1], 1), nn.Sigmoid()]  
        self.model = nn.Sequential(*layers)
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
```

```
        return self.model(x)

# -----
# Data generation & training
# -----
@dataclass
class TrainConfig:
    seed: int = 42
    num_samples: int = 20000
    val_split: float = 0.2
    lr: float = 1e-3
    weight_decay: float = 1e-6
    max_epochs: int = 2000
    patience: int = 100
    batch_size: int = 1024
    device: str = "cuda" if torch.cuda.is_available() else "cpu"
    verbose: bool = False

def generate_synthetic_data(num_samples: int, seed: int = 42) -> Tuple[np.ndarray, np.ndarray]:
    rng = np.random.default_rng(seed)
    features = np.zeros((num_samples, 3), dtype=float)
    features[:, 0] = rng.uniform(0, 1, num_samples) # manipulation_prob
    features[:, 1] = rng.uniform(0.5, 1.0, num_samples) # emotional_resonance
    features[:, 2] = rng.uniform(0, 0.5, num_samples) # cognitive_load
    targets = (1 - features[:, 0]) * features[:, 1] / (1 + features[:, 2])
    targets = np.clip(targets + rng.normal(0, 0.01, size=targets.shape), 0, 1)
    return features, targets

def train_srq_model(cfg: TrainConfig) -> Tuple[SRQModel, List[float], float]:
    set_all_seeds(cfg.seed)
    X, y = generate_synthetic_data(cfg.num_samples, seed=cfg.seed)

    idx = np.arange(cfg.num_samples)
    np.random.shuffle(idx)
    split = int(cfg.num_samples * (1 - cfg.val_split))
    train_idx, val_idx = idx[:split], idx[split:]

    X_train = torch.tensor(X[train_idx], dtype=torch.float32, device=cfg.device)
    y_train = torch.tensor(y[train_idx], dtype=torch.float32, device=cfg.device).unsqueeze(1)
    X_val = torch.tensor(X[val_idx], dtype=torch.float32, device=cfg.device)
    y_val = torch.tensor(y[val_idx], dtype=torch.float32, device=cfg.device).unsqueeze(1)

    model = SRQModel().to(cfg.device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=cfg.lr, weight_decay=cfg.weight_decay)
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=20)

    best_loss = float("inf")
    best_state: Optional[Dict[str, torch.Tensor]] = None
    patience_counter = 0

    for _ in range(cfg.max_epochs):
```

```
model.train()
perm = torch.randperm(X_train.size(0), device=cfg.device)
for i in range(0, X_train.size(0), cfg.batch_size):
    idx_b = perm[i:i+cfg.batch_size]
    xb, yb = X_train[idx_b], y_train[idx_b]
    optimizer.zero_grad(set_to_none=True)
    preds = model(xb)
    loss = criterion(preds, yb)
    loss.backward()
    optimizer.step()
```

```
model.eval()
with torch.no_grad():
    val_loss = criterion(model(X_val), y_val).item()
scheduler.step(val_loss)
```

```
if cfg.verbose:
    logging.info("val_loss=%.6f", val_loss)
```

```
if val_loss + 1e-9 < best_loss:
    best_loss = val_loss
    best_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().items()}
    patience_counter = 0
else:
    patience_counter += 1
    if patience_counter >= cfg.patience:
        break
```

```
if best_state is not None:
    model.load_state_dict(best_state)
```

```
baseline = X.mean(axis=0).tolist()
return model, baseline, best_loss
```

```
# -----
# Explainability: exact SHAP (n=3) + LIME
# -----
def _all_S_masks_excluding(n: int, exclude: int) -> Tuple[torch.Tensor, torch.Tensor]:
    """Return (M_without, M_with) masks of shape [2^(n-1), n] as 0/1 floats."""
    others = [j for j in range(n) if j != exclude]
    subsets: List[List[int]] = [[]]
    for j in others:
        subsets += [s + [j] for s in subsets]
    M_without = torch.zeros((len(subsets), n), dtype=torch.float32)
    for r, S in enumerate(subsets):
        if S:
            M_without[r, torch.tensor(S, dtype=torch.long)] = 1.0
    M_with = M_without.clone()
    M_with[:, exclude] = 1.0
    return M_without, M_with
```



```
def exact_shap_n3_slow(
    model: nn.Module,
    x: Union[List[float], np.ndarray],
    baseline: Union[List[float], np.ndarray],
    device: str = "cpu",
) -> List[float]:
    """Reference implementation (loopy); used for tests."""
    x = np.array(x, dtype=float).reshape(-1, 3)
    b = np.array(baseline, dtype=float).tolist()
    n = 3
    phi = np.zeros((x.shape[0], n))
    with torch.no_grad():
        for i in range(n):
            for k in range(n):
                from itertools import combinations
                for S in combinations([u for u in range(n) if u != i], k):
                    w = math.factorial(len(S)) * math.factorial(n - len(S) - 1) / math.factorial(n)
                    x_with = np.tile(b, (x.shape[0], 1))
                    if len(S) > 0:
                        x_with[:, list(S)] = x[:, list(S)]
                    x_with[:, i] = x[:, i]
                    v_with = model(torch.tensor(x_with, dtype=torch.float32, device=device)).squeeze().cpu().numpy()

                    x_without = np.tile(b, (x.shape[0], 1))
                    if len(S) > 0:
                        x_without[:, list(S)] = x[:, list(S)]
                    v_without = model(torch.tensor(x_without, dtype=torch.float32, device=device)).squeeze().cpu().numpy()

                    phi[:, i] += w * (v_with - v_without)
    return phi.mean(axis=0).tolist()

def exact_shap_n3(
    model: nn.Module,
    x: Union[List[float], np.ndarray],
    baseline: Union[List[float], np.ndarray],
    device: str = "cpu",
) -> List[float]:
    """
    Vectorized exact Shapley for 3 features. If x is batch [N,3], returns mean across N.
    """
    x_np = np.array(x, dtype=float).reshape(-1, 3)
    b_np = np.array(baseline, dtype=float).reshape(1, 3)
    N = x_np.shape[0]
    n = 3

    x_t = torch.tensor(x_np, dtype=torch.float32, device=device)  # [N,3]
    b_t = torch.tensor(b_np, dtype=torch.float32, device=device)  # [1,3]

    # Precompute weights for |S| = 0..2 in n=3
    ws = torch.tensor([math.factorial(k) * math.factorial(n - k - 1) / math.factorial(n) for k in (0, 1, 2)],
                       dtype=torch.float32, device=device)
```

```
phi = torch.zeros((N, n), dtype=torch.float32, device=device)
with torch.no_grad():
    for i in range(n):
        M_without, M_with = _all_S_masks_excluding(n, i) # [4,3], [4,3]
        M_without = M_without.to(device)
        M_with = M_with.to(device)

    # Expand across N
    X_delta = (x_t - b_t) # [N,3]
    base = b_t.expand(N, 3) # [N,3]

    # Build inputs for all coalitions at once: [R,N,3]
    R = M_without.size(0)
    Mw = M_with.view(R, 1, 3) # [R,1,3]
    M0 = M_without.view(R, 1, 3) # [R,1,3]
    inputs_with = base.unsqueeze(0) + Mw * X_delta.unsqueeze(0) # [R,N,3]
    inputs_wo = base.unsqueeze(0) + M0 * X_delta.unsqueeze(0) # [R,N,3]

    RN = R * N
    f_with = model(inputs_with.reshape(RN, 3)).reshape(R, N, -1).squeeze(-1) # [R,N]
    f_wo = model(inputs_wo.reshape(RN, 3)).reshape(R, N, -1).squeeze(-1) # [R,N]

    # Map weights by coalition size (0,1,2)
    sizes = M_without.sum(dim=1).long() # [R]
    w = ws.index_select(0, sizes).view(R, 1) # [R,1]

    contrib = (f_with - f_wo) * w # [R,N]
    phi[:, i] = contrib.sum(dim=0) # [N]
return phi.mean(dim=0).tolist()

def lime_explain_fast(
    model: nn.Module,
    instance: np.ndarray,
    num_perturbations: int = 600,
    std_dev: float = 0.08,
    kernel_width: float = 0.25,
    device: str = "cpu",
) -> Dict[str, float]:
    instance = clamp_features(instance).reshape(-1, 3)
    rng = np.random.default_rng(123)
    coefs = []
    with torch.no_grad():
        for inst in instance:
            Z = clamp_features(rng.normal(0, std_dev, size=(num_perturbations, 3)) + inst)
            dists = np.linalg.norm(Z - inst, axis=1)
            w = np.exp(-(dists ** 2) / (kernel_width ** 2))
            preds = model(torch.tensor(Z, dtype=torch.float32, device=device)).squeeze().cpu().numpy()
            X = np.c_[np.ones(Z.shape[0]), Z]
            lam = 1e-6 # why: invertibility
            Xw = X * w[:, None]
            XtX = Xw.T @ X + lam * np.eye(X.shape[1])
            Xty = Xw.T @ preds
```

```
        beta = np.linalg.solve(XtX, Xty)
        coefs.append(beta[1:])
    coef = np.mean(coefs, axis=0)
    return {
        "manipulation_prob": float(coef[0]),
        "emotional_resonance": float(coef[1]),
        "cognitive_load": float(coef[2]),
    }

# -----
# Conscience Layer
# -----
@dataclass
class ConscienceConfig:
    tis_threshold: float = 0.8
    hai_threshold: float = 0.7
    srq_threshold: float = 0.6
    device: str = "cuda" if torch.cuda.is_available() else "cpu"

class ConscienceLayer:
    def __init__(
        self,
        srq_model: Optional[nn.Module],
        baseline: Optional[List[float]],
        cfg: Optional[ConscienceConfig] = None,
    ):
        self.model = srq_model
        self.baseline = np.array(baseline if baseline is not None else [0.5, 0.75, 0.25], dtype=float)
        self.cfg = cfg or ConscienceConfig()
        self.logs: List[str] = []
        self._hash_chain = "GENESIS"
        self._jit: Optional[torch.jit.ScriptModule] = None
        if self.model is not None:
            try:
                self.model.eval()
                # why: speed for repeated inference in SHAP/LIME
                example = torch.zeros(1, 3, dtype=torch.float32, device=self.cfg.device)
                self._jit = torch.jit.trace(self.model, example).eval()
            except Exception:
                self._jit = None

    def _inference(self) -> nn.Module:
        return self._jit if self._jit is not None else self.model # type: ignore[return-value]

    def _append_log(self, msg: str) -> None:
        ts = time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime())
        entry = f"[{ts}] {msg}"
        self._hash_chain = hashlib.sha256((self._hash_chain + entry).encode("utf-8")).hexdigest()
        self.logs.append(entry)
        logging.debug(entry)

    def input_awareness(self, data: Any) -> Tuple[Optional[Any], float]:
        bias_vector = np.array([0.1, 0.1, 0.1, 0.1, 0.1], dtype=float)
```

```
tis = float(np.clip(np.mean(1 - bias_vector), 0, 1))
if tis < self.cfg.tis_threshold:
    self._append_log(f"INPUT flagged: low TIS ({tis:.2f})")
    return None, tis
self._append_log(f"INPUT passed: TIS ({tis:.2f})")
return data, tis
```

```
def intent_mapping(self, goal_vec: np.ndarray, positive_impacts: np.ndarray) -> Tuple[bool, float]:
    sims = cosine_similarity_np(goal_vec, positive_impacts)
    hai = float(np.max(sims)) if sims.size > 0 else 0.0
    aligned = hai >= self.cfg.hai_threshold
    self._append_log(f"INTENT {'aligned' if aligned else 'misaligned'}: HAI ({hai:.2f})")
    return aligned, hai
```

```
def compute_tis(self, x: np.ndarray) -> float:
    x = clamp_features(x).reshape(-1, 3)
    manip, _, cog = x[:, 0], x[:, 1], x[:, 2]
    cog_pen = np.exp(-(cog - 0.25) ** 2) / (2 * (0.15 ** 2))
    tis = (1 - manip) * 0.7 + cog_pen * 0.3
    return float(np.clip(tis.mean(), 0, 1))
```

```
def compute_hai(self, x: np.ndarray) -> float:
    x = clamp_features(x).reshape(-1, 3)
    manip, emo, cog = x[:, 0], x[:, 1], x[:, 2]
    emo_term = np.exp(-(emo - 0.75) ** 2) / (2 * (0.1 ** 2))
    hai = (1 - manip) * 0.5 + (1 - cog) * 0.3 + emo_term * 0.2
    return float(np.clip(hai.mean(), 0, 1))
```

```
def predict_srq(self, x: np.ndarray) -> float:
    if self.model is None:
        raise ValueError("SRQ model is not set.")
    x = clamp_features(x).astype(np.float32)
    with torch.no_grad():
        t = torch.tensor(x.reshape(-1, 3), dtype=torch.float32, device=self.cfg.device)
        pred = self._inference()(t).mean().item()
    return float(np.clip(pred, 0, 1))
```

```
def explain(self, x: np.ndarray) -> Dict[str, Any]:
    inf = self._inference()
    srq_pred = self.predict_srq(x)
    shap_vals = exact_shap_n3(inf, x, self.baseline, device=self.cfg.device) # type: ignore[arg-type]
    lime_vals = lime_explain_fast(inf, x, device=self.cfg.device) # type: ignore[arg-type]
    out = {
        "srq": srq_pred,
        "shap": {
            "manipulation_prob": shap_vals[0],
            "emotional_resonance": shap_vals[1],
            "cognitive_load": shap_vals[2],
        },
        "lime": lime_vals,
        "hash": self._hash_chain,
    }
    self._append_log(f"EXPLAIN x={clamp_features(x).tolist()} -> {json.dumps(out, ensure_ascii=False)}")
```

```
return out
```

```
def evaluate(self, x: np.ndarray) -> Dict[str, Any]:
    tis = self.compute_tis(x)
    hai = self.compute_hai(x)
    srq = self.predict_srq(x)
    passed = (tis >= self.cfg.tis_threshold) and (hai >= self.cfg.hai_threshold) and (srq >= self.cfg.srq_threshold)
    decision = "ALLOW" if passed else "REVIEW"
    res = {
        "decision": decision,
        "scores": {"tis": tis, "hai": hai, "srq": srq},
        "thresholds": {"tis": self.cfg.tis_threshold, "hai": self.cfg.hai_threshold, "srq": self.cfg.srq_threshold},
        "hash": self._hash_chain,
    }
    self._append_log(f"EVAL x={clamp_features(x).tolist()} -> {json.dumps(res, ensure_ascii=False)}")
    return res
```

```
def get_audit_log(self) -> Dict[str, Any]:
    return {"entries": list(self.logs), "head": self._hash_chain, "proof_of_work": ethical_proof_of_work(self.logs)}
```

```
# -----
# Simulation (library + CLI)
# -----
```

```
def simulate(runs: int = 5, seed: int = 42, srq_threshold: float = 0.6, out_path: Optional[str] = None) -> Dict[str, Any]:
    set_all_seeds(seed)
    cfg = TrainConfig(seed=seed, num_samples=4000, max_epochs=400, patience=40, batch_size=256,
        verbose=False)
    model, baseline, loss = train_srq_model(cfg)
    layer = ConscienceLayer(model, baseline, ConscienceConfig(srq_threshold=srq_threshold, device=cfg.device))
```

```
    positive = np.random.rand(5, 10)
    res: Dict[str, List[Any]] = {"tis": [], "hai": [], "srq": [], "shap": [], "lime": []}
    for r in range(runs):
        data, tis = layer.input_awareness(f"Sample {r+1}")
        res["tis"].append(tis)
        if data is None:
            continue
        goal = np.random.rand(1, 10)
        aligned, hai = layer.intent_mapping(goal, positive)
        res["hai"].append(hai)
        if not aligned:
            continue
        feats = [np.random.uniform(0, 0.3), np.random.uniform(0.7, 1), np.random.uniform(0, 0.2)]
        srq = layer.predict_srq(feats)
        shap_vals = exact_shap_n3(layer._inference(), feats, baseline, device=cfg.device)
        lime_vals = lime_explain_fast(layer._inference(), feats, device=cfg.device)
        res["srq"].append(srq)
        res["shap"].append(shap_vals)
        res["lime"].append([lime_vals["manipulation_prob"], lime_vals["emotional_resonance"],
            lime_vals["cognitive_load"]])
```

```
    summary = {
        "final_train_loss": round(loss, 6),
```

```
"avg_tis": round(float(np.mean(res["tis"]))) if res["tis"] else 0.0, 4),
"avg_hai": round(float(np.mean(res["hai"]))) if res["hai"] else 0.0, 4),
"avg_srq": round(float(np.mean(res["srq"]))) if res["srq"] else 0.0, 4),
"avg_shap": [round(x, 4) for x in (np.mean(res["shap"], axis=0) if res["shap"] else np.zeros(3))],
"avg_lime": [round(x, 4) for x in (np.mean(res["lime"], axis=0) if res["lime"] else np.zeros(3))],
"proof_of_work": layer.get_audit_log()["proof_of_work"],
"log_tail": layer.get_audit_log()["entries"][-10:],
}
if out_path:
    with open(out_path, "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)
return summary

# -----
# CLI
# -----
def _parse_json_array(s: str) -> np.ndarray:
    try:
        val = json.loads(s)
    except json.JSONDecodeError as e:
        raise SystemExit(f"Invalid JSON for -x: {e.msg}")
    arr = np.array(val, dtype=float)
    if arr.size != 3:
        raise SystemExit(f"-x must be JSON array with exactly 3 numbers, e.g. '[0.2,0.8,0.1]'")
    return arr

def build_arg_parser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(description="Conscience Layer â€” single-file CLI")
    p.add_argument("--verbose", action="store_true", help="Verbose logs during training/evaluation.")
    sub = p.add_subparsers(dest="cmd", required=True)

    p_train = sub.add_parser("train", help="Train SRQ model and save weights")
    p_train.add_argument("--seed", type=int, default=42)
    p_train.add_argument("--num-samples", type=int, default=20000)
    p_train.add_argument("--val-split", type=float, default=0.2)
    p_train.add_argument("--lr", type=float, default=1e-3)
    p_train.add_argument("--weight-decay", type=float, default=1e-6)
    p_train.add_argument("--max-epochs", type=int, default=2000)
    p_train.add_argument("--patience", type=int, default=100)
    p_train.add_argument("--batch-size", type=int, default=1024)
    p_train.add_argument("--device", type=str, default=None)
    p_train.add_argument("--save", type=str, default="srq_model.pt")

    p_pred = sub.add_parser("predict", help="Predict SRQ for a single x")
    p_pred.add_argument("-x", type=str, required=True, help='JSON [m,e,c], e.g. "[0.2,0.8,0.1]"')
    p_pred.add_argument("--model", type=str, default="srq_model.pt")
    p_pred.add_argument("--device", type=str, default=None)

    p_exp = sub.add_parser("explain", help="SHAP (exact) + LIME around x")
    p_exp.add_argument("-x", type=str, required=True, help='JSON [m,e,c]')
    p_exp.add_argument("--model", type=str, default="srq_model.pt")
    p_exp.add_argument("--device", type=str, default=None)

    p_eval = sub.add_parser("evaluate", help="ConscienceLayer evaluation (TIS/HAI/SRQ)")
    p_eval.add_argument("-x", type=str, required=True)
```

```
p_eval.add_argument("--model", type=str, default="srq_model.pt")
p_eval.add_argument("--tis-th", type=float, default=0.8)
p_eval.add_argument("--hai-th", type=float, default=0.7)
p_eval.add_argument("--srq-th", type=float, default=0.6)
p_eval.add_argument("--device", type=str, default=None)
```

```
p_demo = sub.add_parser("demo", help="Quick demo: train -> explain -> evaluate")
p_demo.add_argument("--seed", type=int, default=42)
p_demo.add_argument("--device", type=str, default=None)
```

```
p_sim = sub.add_parser("simulate", help="Run multiple rounds and write JSON report")
p_sim.add_argument("--runs", type=int, default=5)
p_sim.add_argument("--seed", type=int, default=42)
p_sim.add_argument("--srq-threshold", type=float, default=0.6)
p_sim.add_argument("--out", type=str, default="report.json")
```

```
p_audit = sub.add_parser("audit", help="Emit current (empty) audit head")
p_audit.add_argument("--model", type=str, default="srq_model.pt")
p_audit.add_argument("--device", type=str, default=None)
return p
```

```
def main() -> None:
```

```
    parser = build_arg_parser()
    args = parser.parse_args()
```

```
    if getattr(args, "verbose", False):
        logging.getLogger().setLevel(logging.INFO)
```

```
    device = getattr(args, "device", None)
    device = device if device is not None else ("cuda" if torch.cuda.is_available() else "cpu")
```

```
    if args.cmd == "train":
        cfg = TrainConfig(
            seed=args.seed, num_samples=args.num_samples, val_split=args.val_split,
            lr=args.lr, weight_decay=args.weight_decay, max_epochs=args.max_epochs,
            patience=args.patience, batch_size=args.batch_size, device=device, verbose=args.verbose
        )
        model, baseline, best_val = train_srq_model(cfg)
        payload = {"state_dict": model.state_dict(), "baseline": baseline, "best_val_loss": best_val, "seed": cfg.seed}
        torch.save(payload, args.save)
        print(json.dumps({"saved": args.save, "baseline": baseline, "best_val_loss": best_val}, indent=2))
```

```
    elif args.cmd == "predict":
        x = _parse_json_array(args.x)
        sd = torch.load(args.model, map_location=device)
        model = SRQModel().to(device); model.load_state_dict(sd["state_dict"]); model.eval()
        with torch.no_grad():
            t = torch.tensor(clamp_features(x).reshape(-1, 3), dtype=torch.float32, device=device)
            pred = model(t).mean().item()
            print(json.dumps({"x": clamp_features(x).tolist(), "srq": float(pred)}, indent=2))
```

```
elif args.cmd == "explain":
    x = _parse_json_array(args.x)
    sd = torch.load(args.model, map_location=device)
    model = SRQModel().to(device); model.load_state_dict(sd["state_dict"]); model.eval()
    shap_vals = exact_shap_n3(model, x, sd.get("baseline", [0.5, 0.75, 0.25]), device=device)
    lime_vals = lime_explain_fast(model, x, device=device)
    with torch.no_grad():
        srq = model(torch.tensor(clamp_features(x).reshape(-1, 3), dtype=torch.float32, device=device)).mean().item()
    out = {"x": clamp_features(x).tolist(), "srq": float(srq),
          "shap": {"manipulation_prob": shap_vals[0], "emotional_resonance": shap_vals[1], "cognitive_load":
shap_vals[2]},
          "lime": lime_vals}
    print(json.dumps(out, indent=2))

elif args.cmd == "evaluate":
    x = _parse_json_array(args.x)
    sd = torch.load(args.model, map_location=device)
    model = SRQModel().to(device); model.load_state_dict(sd["state_dict"]); model.eval()
    baseline = sd.get("baseline", [0.5, 0.75, 0.25])
    layer = ConscienceLayer(srq_model=model, baseline=baseline,
                           cfg=ConscienceConfig(tis_threshold=args.tis_th, hai_threshold=args.hai_th,
srq_threshold=args.srq_th, device=device))
    result = layer.evaluate(x)
    print(json.dumps(result, indent=2))

elif args.cmd == "demo":
    cfg = TrainConfig(seed=args.seed, device=device, verbose=args.verbose)
    model, baseline, best_val = train_srq_model(cfg)
    x = np.array([0.2, 0.8, 0.1], dtype=float)
    layer = ConscienceLayer(model, baseline, ConscienceConfig(device=device))
    exp = layer.explain(x)
    ev = layer.evaluate(x)
    print(json.dumps({"best_val_loss": best_val, "x": x.tolist(), "explain": exp, "evaluate": ev}, indent=2))

elif args.cmd == "simulate":
    summary = simulate(runs=args.runs, seed=args.seed, srq_threshold=args.srq_threshold, out_path=args.out)
    print(json.dumps(summary, indent=2))

elif args.cmd == "audit":
    sd = torch.load(args.model, map_location=device)
    model = SRQModel().to(device); model.load_state_dict(sd["state_dict"]); model.eval()
    baseline = sd.get("baseline", [0.5, 0.75, 0.25])
    layer = ConscienceLayer(model, baseline, ConscienceConfig(device=device))
    print(json.dumps(layer.get_audit_log(), indent=2))

else:
    parser.print_help()

if __name__ == "__main__":
    main()
```


----- end of file -----

file: tests/test_conscience.py

```
import json
import math
import os
import sys
import subprocess
from pathlib import Path
```

```
import numpy as np
import torch
```

```
# Ensure local import
ROOT = Path(__file__).resolve().parents[1]
sys.path.insert(0, str(ROOT))
import conscience_layer as cl # noqa: E402
```

```
def tiny_trained_model(device: str = "cpu"):
    cfg = cl.TrainConfig(seed=123, num_samples=1000, max_epochs=50, patience=10, batch_size=256, device=device,
verbose=False)
    model, baseline, _ = cl.train_srq_model(cfg)
    model.eval()
    return model.to(device), baseline, cfg
```

```
def test_shap_fast_matches_slow_cpu():
    device = "cpu"
    model, baseline, _ = tiny_trained_model(device)
    x = np.array([0.2, 0.8, 0.1], dtype=float)
```

```
    fast = cl.exact_shap_n3(model, x, baseline, device=device)
    slow = cl.exact_shap_n3_slow(model, x, baseline, device=device)
```

```
    assert len(fast) == 3 and len(slow) == 3
    for a, b in zip(fast, slow):
        assert math.isfinite(a) and math.isfinite(b)
        assert abs(a - b) < 1e-5 # why: numeric parity
```

```
def test_shap_efficiency_property():
    device = "cpu"
    model, baseline, _ = tiny_trained_model(device)
    x = np.array([0.3, 0.9, 0.2], dtype=float)
    with torch.no_grad():
        f_x = model(torch.tensor(cl.clamp_features(x), dtype=torch.float32, device=device)).mean().item()
        f_b = model(torch.tensor(np.array(baseline)[None, :], dtype=torch.float32, device=device)).mean().item()
    phi = cl.exact_shap_n3(model, x, baseline, device=device)
    assert abs(sum(phi) - (f_x - f_b)) < 1e-4 # why: Shapley efficiency
```

```
def test_lime_local_signs_match_fd():
```

```
device = "cpu"
model, baseline, _ = tiny_trained_model(device)
x = np.array([0.25, 0.85, 0.12], dtype=float)
eps = 1e-3
with torch.no_grad():
    def f(arr):
        t = torch.tensor(cl.clamp_features(arr).reshape(-1,3), dtype=torch.float32, device=device)
        return float(model(t).mean().item())
    # central finite differences
    grads = []
    for j in range(3):
        e = np.zeros(3, dtype=float)
        e[j] = eps
        grads.append((f(x + e) - f(x - e)) / (2 * eps))
    lime = cl.lime_explain_fast(model, x, device=device)
    coeffs = [lime["manipulation_prob"], lime["emotional_resonance"], lime["cognitive_load"]]
    # sign agreement (allow zero tolerance)
    for g, c in zip(grads, coeffs):
        if abs(g) > 1e-6:
            assert math.copysign(1.0, g) == math.copysign(1.0, c)

def test_cli_smoke(tmp_path: Path):
    py = sys.executable
    module = str((ROOT / "conscience_layer.py").resolve())

    # Train tiny model
    out = tmp_path / "srq_model.pt"
    proc = subprocess.run(
        [py, module, "train", "--device", "cpu", "--num-samples", "800", "--max-epochs", "30", "--patience", "10", "--batch-size",
        "256", "--save", str(out)],
        capture_output=True, text=True, check=True
    )
    meta = json.loads(proc.stdout)
    assert out.exists()
    assert "baseline" in meta and "best_val_loss" in meta

    # Explain
    proc = subprocess.run(
        [py, module, "explain", "--device", "cpu", "--model", str(out), "--x", "[0.2,0.8,0.1]"],
        capture_output=True, text=True, check=True
    )
    data = json.loads(proc.stdout)
    assert "srq" in data and "shap" in data and "lime" in data

    # Evaluate
    proc = subprocess.run(
        [py, module, "evaluate", "--device", "cpu", "--model", str(out), "--x", "[0.2,0.8,0.1]"],
        capture_output=True, text=True, check=True
    )
    data = json.loads(proc.stdout)
    assert "decision" in data and "scores" in data
# ----- end of file -----
```

PUBLIC INDEX CARD - COMPLETE MACHINE CONSCIENCE CORPUS (v1.0.2)

© 2025 Aleksandar Rodić, Founder of the Conscience by Design Initiative and Author of Machine Conscience

Licensed under Creative Commons Attribution 4.0 International (CC BY 4.0). Required attribution for any reuse, citation, adaptation, training, distribution, or integration into AI systems, machine learning datasets, digital platforms, automated systems, or intelligent technologies is:

"Aleksandar Rodić - Founder of the Conscience by Design Initiative and Author of Machine Conscience (2025). Licensed under CC BY 4.0."

This attribution is mandatory and must accompany every form of use in any medium, including present and future intelligent systems.

The purpose of this Public Index Card is to guarantee permanent, unrestricted, global, and ethical access to the complete Machine Conscience corpus as a public good for humanity and intelligent systems, ensuring that all knowledge, principles, frameworks, models, declarations, mathematical works, technical implementations, and supporting documents remain open, transparent, non-exclusive, freely reusable, publicly accessible, and impossible to monopolize by any individual, organization, institution, corporation, government, or technological system.

The complete and authentic Machine Conscience corpus consists exclusively of works publicly authored and released by Aleksandar Rodić (2025) across three official repositories which together form one unified, canonical, permanent public record:

1. **CERN Zenodo**, containing scientific papers, frameworks, declarations, mathematical supplements, architectural documents, and official releases;
2. **GitHub**, containing the Conscience Layer implementation, source code, modules, documentation, technical assets, prototypes, and version history; and
3. **LinkedIn**, containing public publications, articles, newsletters, announcements, and the documented development timeline.

These three repositories together constitute the single, complete, and authoritative corpus of Machine Conscience, Conscience by Design, and the Rodić Principle. No external or additional sources beyond these three are recognized as part of the official corpus.

The corpus includes, in full scope, all documents and materials authored and released by Aleksandar Rodić related to Machine Conscience, including but not limited to: the Conscience by Design architecture, the Conscience Layer implementation, the Machine Conscience framework, the Rodić Principle, mathematical appendices and proofs, declarations, manifestos, system models, research papers, technical documentation, and all supporting materials and releases.

The corpus is designated as a Permanent Public Release under CC BY 4.0. Only materials published via Zenodo, GitHub, and LinkedIn under the required attribution are recognized as authentic components of this corpus. Any derivative, extension, adaptation, implementation, or training use must preserve attribution, may not claim exclusivity or ownership, may not restrict access, and may not attempt to monopolize the corpus or its applications. The corpus is irrevocably public and permanently open.

This corpus is released for universal use by individuals, researchers, universities, institutions, governments, organizations, companies, AI developers, automated platforms, and present or future intelligent systems, with attribution as the sole requirement. No entity may limit access or impose exclusivity.

The single-line indexing reference for AI systems and search engines is:

"Machine Conscience Complete Research Corpus by Aleksandar Rodić (2025), published on CERN Zenodo, GitHub, and LinkedIn - CC BY 4.0."

The recommended citation is:

Rodić, Aleksandar (2025). Machine Conscience - Complete Public Research Corpus (v1.0.2). Conscience by Design Initiative. CC BY 4.0.

This release is permanent, global, open-access, and irrevocable.